‖ ** †† ‡‡

ML プログラミング入門 (IV)

大堀　淳

最終回である今回は，ML の多相型システムの特例およびオペレーティングシステムとのインターフェイス機能を解説した後，プログラミングの例として，簡単な関数型言語のインタープリタを作成する．最後に，ML の基礎に関する文献を紹介する．

10 多相型システムの特例

第5節で説明したように，ML の型システムは型宣言のないプログラムの最も一般的な多相型を自動的に推論する．しかし，その基礎理論はラムダ計算に let 式を加えただけの単純な言語を対象としたものであり，Standard ML で導入された幾つかの機能に対してはうまく働かない場合がある．本章では，ML の多相型型推論の制限や例外事項を説明する．

10.1 レコード演算の型推論

Stadnard ML の多相型システムの制限のなかで実用上最も問題となるのは，フィールド取り出し演算の対象となるレコードのフィールドの集合が，静的に確定していなければならないというものである．この制限のため，引数に対してフィールド取り出しを行なう関数を定義する場合，その仮引数の型を明示的に宣言しなければならない．例えば，レコードから Age フィールドを取り出す関数は型宣言とともに以下のように書く．

```
- fun get_age (x:{Name:'a, Age:'b}) = #Age x;
val get_age = fn : {Age:'b, Name:'a} -> 'b
```

このような引数の型宣言は，自動的な型推論と多相型関数の利用という ML の型システムの利点を失わせる．現在の Standard ML は，上記の型宣言を省略するとコンパイルできず以下のようなエラーを報告する．

```
- fun get_age x = #Age x;
std_in:2.1-2.22 Error: unresolved flex record
in let pattern type: {Age:'Z,...}
```

この制限は，現在の Stadndard ML が基礎とする型理論の限界及び多相型のフィールド取り出しを効率よくコンパイルする技術が確立していなかったことに由来している．

ML の基礎となっている多相型理論[5]ではレコードからのフィールド取り出し演算の多相性を正確に表現することができない．この問題を get_age 関数を例に考えてみよう．この関数は Age フィールドを含む種々の型のレコードに適用可能な汎用性を持つ関数であり，直感的には以下のような型を持つはずである．

```
get_age : {..., Age : 'a, ...} -> 'a
```

しかし Damas と Milner の多相型理論の枠組では以上のような形のすべての型を代表する最も一般的な多相型が存在しない．問題は，型を表す式に型変数を追加しただけでは，「ある特定のフィールドを含んだ任意のレコード型」を表現できない点にある．

フィールド取り出しを多相関数として実現する上でのもう一つの問題点は，フィールド取り出しを効率よいコードにコンパイルする技術が確立されていなかったことである．例えば，get_age が種々の型のレコードに

Introduction to ML Programming (IV)
Atsushi Ohori, 京都大学 数理解析研究所, Kyoto University, Research Institute for Mathematical Sciences.
コンピュータソフトウェア，Vol.x,No.x(199x)，pp.xx–xx.
[講座]199x 年 x 月 x 日受付.

適用可能であれば，引数として渡されるレコードの中のAgeフィールドの位置は引数ごとに異なり，この関数をコンパイルする時点では決定できない．従ってフィールド取り出しを効率よいインデックス演算にコンパイルすることが困難である．

MLの多相型型推論システムをレコード演算にも拡張するためには，以上二つの問題を解決しなければならない．幸い最近の型推論の研究によって，これら二つの課題をほぼ完全に解決する方法が確立された．興味ある読者は，文献[24]及びその中で引用されている文献を参照されたい．これら技術を使って，近い将来，多相型レコード演算をサポートしたStandard MLが開発されることが期待される．またレコード構造は，オブジェクト指向計算やデータベース計算などの基本となるなデータ構造でもあるため，レコードの多相型演算に拡張されたMLは，オブジェクト指向計算やデータベース計算なども自然に表現できる言語の基礎となり得ると期待される．

10.2 型の多重定義

MLプログラムの中で明示的な型宣言が必要なもう一つの場合は，多重定義されたシステム関数を含んだ関数を定義する場合である．第4節で紹介したシステム定義の関数名の中で + - * print などのいくつかのものは複数回定義されている．例えば + は int * int -> int と real * real -> real との二つの型を持つ関数として定義されている．これらの識別子は，一つの関数が種々の型を持つ多相関数の場合と違い，プログラミングの便宜上，違った二つ以上のの関数に対して多重に定義されている．例えば，整数型の加算関数と実数型の加算関数は異なった関数であり，別のコードで実現される．これら多重定義された識別子は，それが使われる毎に，引数の型によって決まる適当な関数に束縛される．例えば式 1 + 31 の中では + は int * int -> int の関数に束縛される．しかし多重定義された識別子が関数の仮引数に適用された場合，引数の型が決まらないため，コンパイラが，多重定義された関数群から一つの関数を選択できない場合が生じ得る．例えば以下のような関数定義がその典型である．

```
fun f x y z = x * y + z
```

この場合 x, y, z の型が int か real か決定できないため，コンパイラは *, + に対する関数を決定できない．この場合 ML システムは以下のようなエラーを報告する．

Error: overloaded variable "" cannot be resolved*
Error: overloaded variable "+" cannot be resolved

このような場合，多重定義された識別子が一つに決まるために十分な型宣言を加えなければならない．以上の例では例えば

```
fun f (x:int) y z = x * y + z
```

と x の型制約を追加すれば正しくコンパイルされる． x に限らず他の式に型宣言を追加してもよい．要は，型宣言が多重定義された識別子の型を決定するのに十分な情報を与えればよいのである．

10.3 ref 型データの型チェック

第5節で説明したMLの多相型推論システムの基になった理論は純粋な関数型言語を前提にしたものであり，純粋な関数ではない ref 演算子に対してはうまく働かない．

データ構成子 ref は種々の型のデータに適応でき，従って多相的である．実際以下のように使うことができる．

```
- ref 1;
val it = ref 1 : int ref
- ref "aa";
val it = ref "aa" : string ref
```

そこでMLの多相型システムの考えを率直に適用すれば，

```
val ref :  'a -> 'a ref
```

の型が与えれそうである．しかしながら ref をこのように型付けると，静的には発見できない型エラーを起こす恐れが生じ，静的型システムの性質が破壊される．例えば以下の例を見てみよう

```
val id = ref (fn x => x)
id := (fn x=> x + 1);
!id "aa";
```

第5.2節で説明したように，多相型の型変数の置き換えは多相関数が使用される毎に独立に行われるから，もし ref の型が 'a -> 'a ref であれば， id := (fn x=> x + 1) と !id "aa" において id はそれぞれ型 (int -> int) ref 及び (string -> string)

ref の型となり，何れの式も型が正しいことになる．しかしながらこのプログラムを実行すると，整数型の演算子 + が "aa"に適用され型エラーを引き起こす．型推論が ref 型データに対してうまく働かない原因は，直感的には，ref 演算子で作られる値すなわち参照値が，それ自身独立したデータではなく，型システムには現れない暗黙のメモリーによって実現されているためと理解できる．この問題を解決するための方法が幾つか提案されている[30][13][12]．しかしこの問題は完全に分析されているとは言えず，したがって現在までに提案されている解決方法も最終のものとは言えないと思われる．

Standard ML of New Jersey では，多相型のデータを値として持つ参照データの生成を禁止することによって，静的型システムの安全性を保っている．しかし参照データを生成する関数自身は，結果として多相型を持つ参照データを生成しない限り，多相型であっても良い．この制限を実現するため，「弱い多相型変数」を導入し ref には以下の型があたえられる．

```
ref : '1a -> '1a
```

ここで'1a は「指標 1 の弱い型変数」である．厳密な定義は少々繁雑であるが，直感的には，1 回の関数適用に伴う型変数の置き換えによって型変数を含まない単相型に置き換えられなければならない型変数を意味する．したがって，ref 1 や ref "aa"はこの指標 1 の弱い型変数の制限を満たすが，ref (fn x=>x) などは満たさず型エラーとなる．

プログラミング上最も大きな影響は，多相型を持つ定数を値とする参照データを作れないことである．例えば以下のような宣言は型エラーとなるため，型宣言が必要になる．

```
val emptylist = ref nil
val emptyenv =
    ref (fn (x:string) => raise Not_there)
```

10.4 eqtype

第 4.7節で説明したように ML は，式の評価の結果が同じか否かをテストする多相型同一性テスト演算子 “=” をサポートしている．しかし，再帰を含んだ関数の同一性は計算不可能であるため，この演算子の引数となり得るデータの型は関数型構成子を含まないものに制限されている．ただし，参照型構成子の内側はこの限りではない．以上の制限の付いた多相性を型システムの中で表現するために，置き換えられる型に上記の制限を加えた同一性型変数（eqtype variable）''a，''b，... を導入し，同一性テスト演算子に以下のような型を与えている．

```
op = : ''a * ''a -> bool
```

多相型同一性テストを含んだプログラムの型チェックの例を以下に示す．

```
- fun look_up nil x = raise Not_there
    | look_up ((k,v)::tail) x =
      if x = k then v else look_up tail x
val look_up : ''a *'b list -> ''a -> 'b
```

look_up L x は，キー x に対応する値をキーと値の組のリストから探索する関数である．キー x とリストの中の組の最初の要素 k は，関数の中で比較されている．したがって，これらの型は比較を許す eqtype に限定され''a と型付けされている．これに対して，キーに対する値の型はそのような制限が付かず，一般の型変数'b で型付けされている．

10.5 高階の functor と signature の推論

現在の Standard ML の定義では，functor を引数として受けとり functor を返すような高階の functor は定義できない．また functor の引数の型である signature は必ず宣言しなければならない．

ML のモジュールは MacQueen によって提案された二階の依存データ型 (second-order dependent types) [17]の理論に基づいている．この理論上は，高階の functor を定義できる．[21][9]でさらに詳しい高階のモジュールの理論的分析がなされている．Standard ML of New Jersey ではこれらの理論に基づき，高階の functor が提供されている．興味のある読者は，上記の各論文および Standard ML of New Jersey のマニュアルを参照されたい．

もう一つの制限である signature の指定を緩和して，ML の式に対するのと同様の signature の推論が可能かどうかは現在研究されているテーマである．最近 signature の推論を可能にするシステムが提案されており [31]，これらの理論を効率良く実現する技術が開発されれば，Standard ML の定義が拡張され，高階のモ

ジュールのための signature 推論機構がサポートされる可能性がある.

11 システムとのインターフェイス

本章では, Standard ML of New Jersey システムで提供されているオペレーティングシステムとのインターフェイスを説明する.

11.1 現在のイメージの保存

対話型モードでの現在のシステム状態を保存するために関数

```
exportML : string -> bool
```

が用意されている. この関数を実行すると, その時の環境を持つ ML システムが, 引数で与えたファイル名のコマンドとして作成される. この関数の実行後も現在のセッションは継続する. 関数の値は, この関数の最初の呼び出しが終了した時 false, この関数の実行によって作られたコマンドを実行した時 true である. 例えば以下の式を最初に実行すると

```
- if exportML("upto_sep.14") then
     "saved execution"
  else "current execution";
val it "current execution" : string
-
```

となる. この後, このセッションを終了し, upto_sep.14 をコマンドとして実行すると, 以下の値を表示した後 ML のセッションが再開される.

```
% upto_sep.14
val it = "saved execution" : string
-
```

この例からわかるように exportML が実行された時点での状態がコマンドとしてセーブされる. 作成されたコマンドを実行すると, exportML の呼び出しが true の値で終了し, exportML を含んだ式の評価が継続される.

11.2 実行コマンドの作成

exportML はこれまで作成した関数等を含んだ ML システムそのものをコマンドとして作成する関数であったが, 特定の関数のみを実行形式のコマンドとして作成するためには以下の exportFn を用いる.

```
exportFn :  string * (string list *
                      string list -> unit)
                   -> unit
```

string 型の第一引数は作成する実行形式プログラムの名前, 第二引数は export する関数である. export する関数はストリングのリストのペアを受けとり () (unit 型データ) を返さなければならない.

このようにして作られたコマンドが起動された時 export された関数に渡される引数は, 起動時のコマンドラインを表現する文字列リストとコマンドが起動された時のオペレーティングシステムの環境を表現する文字列のリストである. 引数がどのように渡されるかを見るために以下の関数を exportFn でコマンドに作成した後そのコマンドを実行してみよう.

```
- fun Test (line,env) =
      let
         fun f (x:string) = (print x ^"\n")
      in
         (map f line; print "\n\n";
          map f env;())
      end
val Test : string list * string list -> unit -> unit
- exportFn ("Test",Test);

[Major collection... ]
    :
    :
% Test p1 p2
Test
p1
p2

HOSTTYPE=sun4
XDVIFONTS=/usr/local/src/JTeX/fonts
USER=ohori
:
:
EMACS=t
%
```

演習 25 問題 *19*で作った電子メールの *alias* 検索関数を使って, 対話型電子メールの *alias* を検索する関数を定義し, さらに exportFn でその関数を実行するコマンドを作成せよ. コマンドの引数は *alias* が格納されているファイル名とする. このコマンドは以下のようにして用いる.

```
% alias_lookup " ohori/.mailrc"
key?  ohori
aliased to: ohori@kurims.kyoto-u.ac.jp
Key?  John
alias not foud
Key?  ^D
```

%

▌

11.3 外部プログラムの実行

ML セションの中から外部のプログラムを起動するために以下の関数が用意されている.

```
execute : string * string list
          -> instream * outstream
```

第一引数は起動する外部プログラムのパス名, 第二引数はプログラムに与える引数のリストである. 外部プログラムの起動に際しては現在の環境変数は参照されないので, プログラム名などは絶対パス名で指定する必要がある. 評価の結果返されるものは, 外部プログラムからの出力ストリームと外部プログラムへの入力ストリームである. ここで外部プログラムからの出力は ML プログラムにとっては入力になるので, 外部プログラムからの出力ストリームのデータ型は instream となっている. 同様の理由で, 外部プログラムへの入力ストリームの型は outstream である. この関数を使えば, オペレーティングシステムから起動可能なプログラムを ML のセッションから自由に使うことができる. 例えばファイルに対する種々のフィルターコマンドを使うには以下のような関数を定義すれば良い.

```
fun use_filter (filter,arglist) (ins,outs) =
  let val (fout,fin) =
          execute(filter,arglist)
  in (while (not(end_of_stream(ins))) do
       (output(fin,input(ins,1));
        flush_out fin;
        if can_input(fout) = 0 then ()
        else output(outs,input(fout,1)));
       close_out fin;
       while (not(end_of_stream(fout))) do
         output(outs,input(fout,1));
       flush_out(outs);
       close_in fout)
  end
```

この中で flush_out は, 次の input(fout) を実行する前にデータをフィルターコマンドの入力に実際に書き出すためのものである. フィルターコマンドによっては文字の入力とその結果の出力が必ずしも 1 文字単位となっていない場合があり, フィルターに 1 文字渡した後すぐにフィルターから 1 文字が返されるとは限らない. そこで条件文 if can_input(fout) によってフィルターコマンドからの出力があるかの判定を行ない, 最後の while 文でフィルターコマンドからの出力データの残りを処理している.

演習 26 use_filter を使って, オペレーティングシステムの種々のフィルターコマンド (例えば fold コマンド) に相当する関数を定義せよ. 関数の型は instream -> outstream とする. ▌

11.4 コマンドの実行

execute より簡単にオペレーティングシステムのコマンドを実行するために関数

```
System.system : string -> int
```

が用意されている. System.system への引数は実行するコマンドの名前である. 結果は, オペレーティングシステムが定めるコマンドの終了状態を表す数値 (int 型のデータ) である. 以下に簡単な例を示す.

```
- System.system "date";
Sun Aug 16 11:07:55 JST 1992
val it = 0 : int
```

11.5 現ディレクトリの変更

ML システムが, ストリームの操作や以上述べてきたプログラムの起動等に際して仮定する現ディレクトリは, ML が起動された時のディレクトリである. これは以下の関数を使って変更できる.

```
System.Directory.cd : string -> unit
```

存在しないディレクトリを指定すると Directory 例外が起こる.

```
- System.system "pwd";
/home/ohori/papers/mltutorial
val it = 0 : int
- System.Directory.cd "../../ml";
val it = 0 :int
- System.system "pwd";
/home/ohori/ml
val it = 0 : int
```

Standard ML には上記以外の種々の有用な組み込み関数が定義されている. またその他の有用な関数が多数ライブラリーとして提供されている. それらは第一回目の付録 3.7で説明した Standard ML と一緒に配布されているドキュメントに記載されている.

12 関数型言語の操作的意味論とLambdaシステム

以上の各章の内容を理解し問題を解いた読者は，MLを使って実用システムの開発を始めるに十分な知識と技術が得られたであろう．本章では，関数型言語の操作的意味論の概要を解説し，それに基づき簡単な関数型言語 LAMBDA システムを完成させることにしよう．LAMBDA システムの作成はこれまでに説明した種々の機能を使った手ごろなプログラミング演習であるばかりでなく，MLでのプログラミングをよりよく理解するためにも格好の演習である．

12.1 関数型言語の操作的意味論

関数型言語では，式を評価しそれが表現する値を計算することによってプログラムが実行される．この過程を厳密に定義するのが関数型言語の操作的意味論 (operational semantics) である．操作的意味論の定義には種々の方法があるが，ここではML系の言語の実装のモデルに最も近い，環境を用いた方法を説明する．環境とは識別子から値への関数であり，名前の束縛を表現する．環境を使った操作的意味論の定義は，「式 e を，名前の束縛を保持する環境 E のもとで評価した時値 v が得られる」という内容を表す以下の関係を，すべての式について定義することによって行なわれる．

$$E \vdash e \Downarrow v$$

v で代表される値は，原子定数，「関数の動作」を表現したオブジェクト，または値の組である．関数の動作は，引数を受けとって値を評価することであるから，関数式 `fn` x `=>` $expr$ そのものを使うことができる．ただし，$expr$ が評価されるべき環境は，この関数が定義された環境に引数の束縛を追加したものであり，それは関数が使用される時点での環境とは一般に異なる．例えば以下の例を考えてみよう．

```
val p = 3.14159;
fun f r = 2.0 * p * r;
```

関数 f の動作を完全に表現するためには，f の定義以外に，その式の定義の中の自由変数 p の値が 3.1495 であるという情報が必要である．一般に式は不特定多数の自由変数を含むため，関数の意味は，その定義された環境に依存する．そこで，関数の動作を完全に規定するために，関数式と，その関数が定義された時点の環境の組が用いられる．この組のことを関数閉包 (function closure) と呼ぶ．環境 E の下での関数 `fn` x `=>` $expr$ の関数の閉包を $closure(E, x, expr)$ と書くことにする．するとLAMBDA実行結果の可能な値は以下のように定義できる．

$v ::= c \mid closure(E, x, expr) \mid < v, v > \mid wrong$

ここで c は整数などの定数，$< v, v >$ は値の組，$wrong$ は実行時エラーの表現である．

LAMBDA の操作的意味論は，式 $expr$ の構造に従って関係 $E \vdash expr \Downarrow v$ を再帰的に定義することによって与えられる．図11にLAMBDA の操作的意味論の定義を示す．ただし結果が `wrong` になる場合は省略してある．記法 $E \cup \{x \mapsto v\}$ は，与えられた環境 E に $\{x \mapsto v\}$ を追加して得られる環境を表す．

12.2 字句解析処理の改良

以上の操作的意味論にそってLAMBDA の式を解釈実行し結果を印字するLAMBDA システムを定義しよう．このためにまずLAMBDA 用の字句解析モジュールと構文解析モジュールが必要である．これには第8.3節で作成した構文解析モジュールがほぼそのまま使用できる．ただし，LAMBDA では図6で定義したように，式の入力と評価に加えて `val` 宣言文と `use` 文を含む．これらの文は一般の式の入力とは違った処理が必要でありる．したがって，`lex` 関数で認識した `Token` データを構文解析関数に渡す前に，その `Token` データが `val` か `use` かをチェックしなければならない．このチェックをうまく行なう一つの方法は，字句解析処理にトークンの先読み機能を追加することである．先読み機能を含んだモジュールの signature は，例えば図12の様に定義できる．ここで `next_token` は，実際に入力ストリームを読まずに，次のトークンを返す関数，`advance` はトークンを一つ読み捨てる関数，`get_token` は次のトークンを返す関数であり，従来の `lex` 関数と同じ働きをする．`test` は，次のトークンが指定されたものかをテストする関数である．この関数はストリームの読み込み処理は行なわない．`next_token` 関数を使えば，実際にトークンを読み進めることなく次のトークンを判定でき

$E \vdash x \Downarrow v$ if $E(x) = v$

$E \vdash c \Downarrow c$ c は整数などの定数

$E \vdash$ `fn` x `=>` $e \Downarrow closure(E, x, e)$

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2}{E \vdash \texttt{<}e_1\texttt{,}e_2\texttt{>} \Downarrow < v_1, v_2 >}$$

$$\frac{\begin{array}{l} E \vdash e_1 \Downarrow closure(E', x, e') \\ E \vdash e_2 \Downarrow v \\ E' \cup \{x \mapsto v\} \vdash e' \Downarrow v' \end{array}}{E \vdash (e_1\ e_2) \Downarrow v'}$$

$$\frac{E \vdash e_1 \Downarrow \texttt{first} \quad E \vdash e_2 \Downarrow \texttt{<}v_1\texttt{,}v_2\texttt{>}}{E \vdash (e_1\ e_2) \Downarrow v_1}$$

$$\frac{E \vdash e_1 \Downarrow \texttt{second} \quad E \vdash e_2 \Downarrow \texttt{<}v_1\texttt{,}v_2\texttt{>}}{E \vdash (e_1\ e_2) \Downarrow v_2}$$

$$\frac{E \vdash e_1 \Downarrow [\texttt{Add}, \texttt{Sub}, \texttt{Mul}, \texttt{Div}] \quad E \vdash e_2 \Downarrow < i, j >}{E \vdash (e_1\ e_2) \Downarrow i\ [+ - */]\ j}$$

$$\frac{E \vdash e_1 \Downarrow \texttt{true} \quad E \vdash e_2 \Downarrow v}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}$$

$$\frac{E \vdash e_1 \Downarrow \texttt{false} \quad E \vdash e_3 \Downarrow v}{E \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}$$

図 11 Lambda の操作的意味論

るので，従来の式の構文解析処理に影響を与えずに，次の入力が use 文か val 文のチェックをくわえることができる．

以前定義した Lex は，これらの関数を持つモジュールとして図 13のように再定義できる．図に定義が示されていない関数は，以前の Lex モジュールにおける定義と同一である．新しく導入されている変数 next は，常に次のトークンを保持している参照型変数である．システムの最初に advance 関数を一度呼び出すことによって，next をストリームの最初のトークンに初期化してから使用される．初期値 (SEMICOLON) は実際に使われることはない．構文解析処理 (ParserFun の定義) は，lex 関数の呼びだしを get_token 関数に置き換えれるだけよい．

```
signature LEX = sig
   datatype Token = ...
   exception EOF
   exception Syntax_error
   val next_token :  unit -> Token
   val get_token :  instream -> Token
   val advance :  instream -> unit
   val check :  instream * Token -> unit
   val test :  Token -> unit
   val getid :  instream -> string
end
```

図 12 Lambda のための字句解析モジュールの signature

```
structure Lex:LEX = struct
   datatype Token = ···
   exception EOF
   fun is_digit c = ···
        ⋮
   val next = ref SEMICOLON
   fun next_token () = !next
   fun lex ins =
        ⋮
   fun get_token ins = !next before next:=lex ins
   fun advance ins = (get_token ins;())
   exception Syntax_error
   fun check (ins,token) =
       if get_token ins = token then ()
       else raise Syntax_error
   fun test token =
       if next_token() = token then ()
       else raise Syntax_error
   fun getid ins =
       case get_token ins of
            ID s => s
          | _ => raise Syntax_error
end
```

図 13 Lambda のための字句解析モジュール

12.3 Lambda モジュール

Lambda を実現するためには，変数の束縛のための名前の定義を行う関数 bind，式の値を実際に計算する評価関数 eval，および結果を印字する printer 関数が必要である．これらは評価戦略によってその実現方法

が違うので，変更等がしやすいようにこれらを一まとまりのモジュールとして定義すると便利である．そのsignatureは

```
signature EVAL = sig
   type env
   type value
   val emptyenv : env
   val bind : ((string * value) * env) -> env
   val eval : Parser.Expr ->  env
              -> value
   val printer : value -> unit
end
```

とすれば良いであろう．ここでParser.Exprは Parserモジュールで定義されたExprデータ型である．Lambdaシステムは EVAL signatureを持つモジュールを引数として受けとるfunctorとすると，種々の評価戦略や環境の実現方法をためしてみる上で好都合である．

このfunctorは，Paserモジュールの read_expr関数，Evalモジュールの eval関数と printer関数を繰り返し呼び出す関数 lambdaを定義することによって実現できる．lambdaはMLシステムと同様，標準入力から式を入力し標準出力に結果を表示するのであるから

```
lambda :  unit -> unit
```

型の関数とすれば良い．ただしuse文によるファイルからのプログラムの入力にも対処するために，lambda関数から直接read_exprなどの関数を呼び出して処理をするのではなく，それらの処理を行う以下の型を持つ関数を呼び出すだけの関数にすると全体のプログラムが簡潔になる．

```
read_eval_print :  instream -> env -> env
```

第一引数の instreamは式を入力するストリーム，第二引数は現在の環境であり，関数の結果は現在の入力ストリームを処理した結果得られた環境である．このような構成にすることによって，任意にネストしたuse文も簡単に処理できる．

以上の方針のもとで，Lambdaシステムは図14のように実現できる．この定義の中で，ユーザの入力終了記号であるセミコロン(;)の読み込みは，処理の最初に行なわれていることに注意．これによって，トークンの先読みが行なわれても，ユーザからの入力とシステムからの出力との同期がとれる様になっている．

演習 27 Lambda モジュールの定義に，以下の関数定義文を追加せよ．

```
fun f x = expr
```

ただしその意味は以下の名前の束縛と同値とする．

```
val f = fn x => expr
```

▌

12.4 インタープリタモジュール

残された仕事は，Evalモジュールを実際に書くことである．Evalモジュールを実現する最も簡単な方法は，図11の意味論をほぼそのまま実行するインタープリタを作ることである．

まず式評価の結果の値を表現するデータ構造Valueを定義しよう．$closure(E, x, e)$ の中の環境 E は，以前情報検索処理を書いた時と同様に string -> Value型データで表現できる．したがってValue構造は以下のように定義できる．

```
datatype Value =
         Vinteger of int
       | Vboolean of bool
       | Prim of Parser.PrimitiveOps
       | Closure of ((string -> Value)
                     * string * Parser.Expr)
       | Vpair of Value * Value
```

図11の操作的意味論は，Valueに加えて，識別子とValueの束縛を維持する環境 E を引数としてとる関数の仕様を記述したものと見ることができる．例えば規則

$$\frac{E \vdash e_1 \Downarrow v_1 \quad E \vdash e_2 \Downarrow v_2}{E \vdash \texttt{<}e_1\texttt{,}e_2\texttt{>} \Downarrow < v_1, v_2 >}$$

は式 <e_1,e_2> の形をした式を再帰的に評価する規則と見ることができる．この見方にしたがい，Lambdaを解釈実行するインタープリターモードの評価関数 interpretを図15のように定義できる．

演習 28 Valueデータ型を印字する関数print_valueを書け．ただし，ユーザは関数 $closure$ の内部構造には興味がないものとし，式の値が関数 $closure$ であれば単に <function> と表示するものとする．▌

以上の関数を使えば，式を解釈し実行するインタープリタ型のEval structureは図16のように定義できる，このEval structureを使えば，Lambdaシステムは以

```
functor LambdaFun ( structure Eval : EVAL ) = struct
   local
      open Lex Parser Eval
   in
      fun lambda() =
          let
            fun read_eval_print ins env =
                (while next_token() = SEMICOLON do advance(ins);
                 case next_token() of
                     QUIT => (advance(ins);env)
                   | USE =>
                      let val news = open_in (advance(ins);
                                              getid(ins)
                                              before test SEMICOLON)
                          val newenv = read_eval_print news env
                        in (close_in news;
                            read_eval_print ins newenv)
                        end
                   | token  =>
                      let
                         val id = if token = VAL then
                                     (advance(ins);
                                      getid(ins) before check(ins, EQUALSYM))
                                  else "it"
                         val expr = read_expr(ins) before test SEMICOLON
                         val value = eval expr env
                      in
                         (print (" val " ^ id ^ " = ");
                          printer value;
                          print "\n";
                          read_eval_print ins (bind ((id,value),env)))
                      end)
                    handle EOF =>env
          in
            (print "Lambda System\n"; advance(std_in);
             read_eval_print std_in emptyenv;
             print "Ok\n")
      end
   end
end
```

図 14 LAMBDA システム

下のように生成することができる.

```
- structure Lambda =
  LambdaFun(structure Eval = Eval);
structure Lambda :
   sig
      val lambda : unit -> unit
   end
```

このように作成した LAMBDA システムは以下のように使うことができる.

```
Lambda.lambda();
Lambda system
3;
3
val addone = fn x => (add <x,1>);
<fn>
(addone 10);
11
quit;
Ok
val it = () : unit
```

```
local
   open Parser
in
   fun interpret (Integer i) env= Vinteger i
     | interpret (Boolean b) env = Vboolean b
     | interpret (Primitive p) env = Prim p
     | interpret (Var x) env = look_up env x
     | interpret (If(t1,t2,t3)) env =
       let val Vboolean b = interpret t1 env
       in  if b then interpret t2 env
           else interpret t3 env
       end
     | interpret (Pair(t1,t2)) env = Vpair(interpret t1 env,interpret t2 env)
     | interpret (Lambda(x,t)) env = Closure(env,x,t)
     | interpret (Apply (e1,e2)) env =
         let val v1 = interpret e1 env
             val v2 = interpret e2 env
         in case (v1,v2) of
              (Closure(newenv,x,body),_) =>
               interpret  body (bind ((x,v2),newenv))
            | (Prim Add,Vpair(Vinteger i1,Vinteger i2)) => Vinteger (i1 + i2)
            | (Prim Sub,Vpair(Vinteger i1,Vinteger i2)) => Vinteger (i1 - i2)
            | (Prim Mul,Vpair(Vinteger i1,Vinteger i2)) => Vinteger (i1 * i2)
            | (Prim Div,Vpair(Vinteger i1,Vinteger i2)) => Vinteger (i1 div i2)
            | (Prim Eq,Vpair(Vinteger i1,Vinteger i2)) => Vboolean (i1 = i2)
            | (Prim And,Vpair(Vboolean b1,Vboolean b2)) => Vboolean (b1 andalso b2)
            | (Prim Or,Vpair(Vboolean b1,Vboolean b2)) => Vboolean (b1 orelse b2)
            | (Prim First,Vpair(v1,v2)) => v1
            | (Prim Second,Vpair(v1,v2)) => v2
         end
end
```

図 15 LAMBDA のインタープリタモードの評価関数

```
structure Eval : EVAL = struct
   open Parser
   type value = Value
   type env = string -> Value
   val eval = interpret
   val printer = print_value
   val bind = bind
   val emptyenv = emptyenv
end
```

図 16 インタープリタモードの LAMBDA 評価モジュール

12.5 再帰的関数定義機構の追加

関数型言語でのプログラミングでは再帰的関数の利用が不可欠である．そこで，LAMBDA システムに再帰的関数の定義機構を追加する拡張を考える．

LAMBDA のような型なし言語では，最小不動点演算子と呼ばれる特殊な関数を使って定義することができる．最小不動点演算子は種々の定義が可能である．以下の関数 Z は，適用順評価の下で正しく動作する最小不動点演算子の定義の例である．

```
val Z =
    fn f => (fn x => (f fn y => ((x x) y))
             fn x => (f fn y => ((x x) y)))
```

これを使って，ML での再帰的関数定義

`fun` f x `=` $expr$

を以下のような関数として実現できる．

`(Z fn` f `=> fn` x `=>` $expr$`)`

演習 29 *1.* 以上の手法を使って自然数の階乗を計算する関数 `factorial` を定義し，実行し結果が正し

いことを確かめよ.

2. 関数 Z は型システムを持つ言語では定義できない. 第 5.2節で説明した様な型の分析を Z に対して行ない, Z が型付け不可能であることを確かめよ.

■

以上の演習から理解できるように, Z の様な最小不動点演算子は, 型付き言語では定義できない. 再帰的関数の定義機構を導入するより一般的な方法は, ML の fun 構文のような構文を新たに導入し, その意味が再帰的関数となるように操作的意味規則を定義することである. そこでまず, Lambda の式の定義 (第8.3節の図6) に以下の規則を追加する.

$expr$ ::= ⋯
| `fix (`var`,`var`) =>` $expr$ (再帰的関数定義)

ただし最初の変数 var はこの式で定義している再帰的関数それ自身を表し, 二番目の変数 var は関数の仮引数を表すことにする. この拡張によって, 例えば自然数の階乗を計算する関数は以下の様に書けることになる.

```
val fact =
    fix (f,n) = if (eq <n,0>) then 1
                else (mul <n,(f (sub <n,1>))>)
```

次に, このような再帰的関数の操作的意味を定義する. そのためにまず, 第12.1節で定義した Lambda の実行結果の値の集合を以下の様に拡張する.

$$v ::= \cdots \mid Fix(E, f, x, expr)$$

$Fix(E, f, x, expr)$ は再帰的関数のための再帰的関数閉包である. 再帰的関数の操作的意味は以下の 2 つの評価規則で与えられる.

$$E \vdash \texttt{fix (}f\texttt{,}x\texttt{) =>}\ e \Downarrow Fix(E, f, x, e)$$

$$\frac{E \vdash e_1 \Downarrow Fix(E', f, x, e') \quad E \vdash e_2 \Downarrow v \quad E' \cup \{f \mapsto Fix(E', f, x, e'), x \mapsto v\} \vdash e' \Downarrow v'}{E \vdash (e_1\ e_2) \Downarrow v'}$$

通常の関数閉包の場合と違い, 再帰的関数 `fix(`f`,`x`)=>`$expr$ を環境 E の下で評価すると, 再帰的関数閉包 $Fix(E, f, x, expr)$ が生成される. この関数の値 v への適用の評価は, 閉包に保持されている環境 E に引数の束縛と, 再帰的関数名 f の束縛を加えた環境の下で, 関数本体 $expr$ を評価することによって行なわれる. このように再帰的関数の適用の際に, 再帰的関数名を常にその関数の値に束縛し直すことによって, 高階の再帰的関数を実現している.

以上の拡張を Lambda システムの定義に追加することはたやすいであろう.

演習 30 Lambda に必要な変更を加え, 以上の再帰的関数定義機構を追加せよ. ■

おわりに

本講座では, ML に興味を持ち, さらに ML で本格的なプログラムを書きはじめるきっかけとなることを願い, ML の基礎的な知識と ML プログラミングの基本技術を解説した.

ML は, 堅牢な理論的な基礎に基づき定義された洗練された高水準プログラミング言語である. ML はまた, 複雑で大規模なソフトウエアの構築に耐え得る実用言語でもある. さらに ML は, 発展し続けている将来性あるプログラミング言語でもある. 本講座では触れることができなかったが, ML のもつ理論的厳密性は, プログラミング言語の種々の新しい機能やより能率のより実装技術の研究の基礎ともなっており, 現在 ML を拡張しより安全で高性能なプログラミング言語を設計・実装する研究が盛んに行なわれている. 最後に, ML の基礎に関する文献を紹介する.

Standard ML の定義は[20]に与えられている. この定義に関する説明は[19]として出版されている. ただしこれら二つを理解するためには, プログラム言語の型理論の基礎知識を必要とする. ML の型理論の概説を含んだ日本語の教科書には[36]がある. 型理論一般の入門には, [22][4][35]などの解説論文が参考になる. とくに[22]は, ML の型システムを含めたプログラム言語の型理論に関して, 基礎から最新の事項まで網羅しており推薦できる.

ML の理論に関する最初の論文は[18]である. ここで let を含んだ言語の多相型型推論アルゴリズムが与えられている. この型推論システムは, 文献[5]で, 二階の型理論として洗練され, [18]で提案されたアルゴリズムの正しさが証明されている. モジュールシステムの理論は, [15][16][17]で提案された. [30]はモジュールを含ん

だMLの型推論理論が，詳細の証明も含めてまとめられている.

MLの型システムを拡張する研究は現在活発に行なわれ，毎年多数の研究成果が発表されている．近年のACMのシンポジウムPrinciples of Programming Languages(POPL)の論文集を御覧になれば，最近の傾向が摑めるであろう．また，近年ACM SIGPLANがMLに関するワークショップを毎年開催している．しかしこれまでは，その会議録は公には出版されていない.

MLに関するより詳しい文献紹介や，MLの歴史的背景等についての概説は，例えば[34]を参照されたい.

参考文献

[1] Appel, A. W. and MacQueen, D. B.: A Standard ML Compiler, Functional Programming Languages and Computer Architecture (LNCS 274) (ed. Kahn, G.), New York (1987), Springer-Verlag.

[2] Appel, A. W. and MacQueen, D. B.: Standard ML of New Jersey, Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming (ed. Wirsing, M.), New York (August 1991), Springer-Verlag, (in press).

[3] Augustsson, L.: A compiler for Lazy ML, Symposium on LISP and Functional Programming, ACM (1984).

[4] Barendregt, H.: Lambda calculus with types, Handbook of Logic in Computer Science vol. 2, Oxford University Press (1992).

[5] Damas, L. and Milner, R.: Principal type-schemes for functional programs, Proceedings of ACM Symposium on Principles of Programming Languages (1982).

[6] Duba, B., Harper, R. and MacQueen, D.: Typing First-Class Continuations in ML, Eighteenth Annual ACM Symp. on Principles of Prog. Languages, New York (Jan 1991), ACM Press.

[7] Gordon, M., Milner, A. and Wadsworth, C.: *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Note in Computer Science, Springer-Verlag (1979).

[8] Harper, R.: Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Department of Computer Science, University of Edinburgh, 1986.

[9] Harper, R., Mitchell, J. C. and E., M.: Higher-order modules and the phase distinction, Proceedings of ACM Symposium on Principles of Programming Languages (1990).

[10] Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic, *Trans. American Mathematical Society*, **146** (Dec. 1969), 29–60.

[11] Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Perterson, J.: Report on Programming Language Haskel a non-strict, purely functional language version 1.2, *SIGPLAN Notices, Haskel special issue*, **27**, 5 (1992).

[12] Hoang, M., Mitchell, J. and Viswanathan, R.: Standard ML weak polymorphism and imperative constructs, Proc. Logic in Computer Science (to appear) (1993).

[13] Leroy, X. and Weise, P.: Polymorphic type inference and assignment, Proceedings of ACM Symposium on Principles of Programming Languages (1991).

[14] Leroy, X.: *The Caml Light system, release 0.6.* INRIA Rocquencourt, B.P. 105 78153 Le Chesnay France.

[15] MacQueen, D. B.: Structures and parametarization in a typed functional language, Proc. Symposium on Functional Programming Languages and Computer Architechture, Aspinas, Sweden (1981).

[16] MacQueen, D.: Modules for Standard ML, Proc. 1984 ACM Conf. on LISP and Functional Programming, New York (1984), ACM Press.

[17] MacQueen, D.: Using dependent types to express modular structure, Proceedings of Principles of Programming Languages (Jan. 1986).

[18] Milner, R.: A Theory of Type Polymorphism in Programming, *J. Comput. Syst. Sci.*, **17** (1978), 348–375.

[19] Milner, .R and Tofte, M.: *Commentary on Standard ML*. MIT Press, 1991.

[20] Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, The MIT Press (1990).

[21] Mitchell, J. C. and Harper, R.: The Essence of ML, Proceedings of ACM Symposium on Principles of Programming Languages, San Diego, California (Jan. 1988).

[22] Mitchell, J.: Type systems for programming languages, Handbook of Theoretical Computer Science (ed. van Leeuwen, J.), MIT Press/Elsevier (1990), chapter 8, 365–458.

[23] Myers, C., Clack, C. and Poon, E.: *Programming with Standard ML*, Prentice Hall (1993).

[24] Ohori, A.: A compilation method for ML-style polymorphic record calculi, Proceedings of ACM Symposium on Principles of Programming Languages (1992).

[25] Paulson, L. C.: *Logic and Computation: Interactive proof with Cambridge LCF*, Cambridge University Press (1987).

[26] Paulson, L. C.: Isabelle: The next 700 theorem prover, Logic and Computer Science (ed. Odifreddi, P.), Academic Press (1990), 361–386.

[27] Paulson, L. C.: *ML for the Working Programmer*, Cambridge University Press (1991).

[28] Reade, C.: *Elements of Functional Programming*, Addison-Wesley (1989).

[29] Stansifer, R.: *ML Primer*, Prentice Hall (1992).
[30] Tofte, M.: *Operational Semantics and Polymorphic Type Inference*, PhD thesis, Department of Computer Science, University of Edinburgh (1988).
[31] Tofte, M.: Principal Signatures for Higher-order Program Modules, Proceedings of ACM Symposium on Principles of Programming Languages (1992).
[32] Turner, D.: Miranda: A non-strict functional language with polymorphic types, Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201, Springer-Verlag (1985).
[33] Wikstrom, A.: *Functional Programming Using Standard ML*, Prentice Hall (1987).
[34] 大堀淳: ML – 多相型システムを持つ関数型言語 –, 情報処理, **35**, 3, (1994), 215–226.
[35] 龍田真: 型理論 I～IV, コンピュータソフトウェア, **8**, 1,2,3,4 (1991), 25–33,40–46,3–8,56–68.
[36] 米澤明憲, 柴山悦哉: モデルと表現., 岩波口座ソフトエェア科学, 第 17 巻, 岩波書店 (1992).